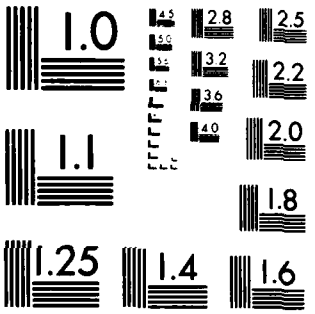
AD-A144 385 THE GRID FILE: A DATA STRUCTURE DESIGNED TO SUPPORT PROXIMITY QUERIES ON..(U) INSTITUT FUER INFORMATIK ZURICH (SWITZERLAND) K HINRICHS ET AL. JUN 83 1/1

UNCLASSIFIED DAJA37-82-C-0058 F/G 12/1 NL

END
DATE FILMED
9 84
DTIC



(1)

Annual Progress Report #1, June 1983

ERO Contract No. DAJA 37-82-C-0058

Survey of Efficient Data and File Structures

Principal investigator:
Prof. Jurg Nievergelt, Informatik, ETH, CH-8092 Zurich

The first-year activities supported by this contract fell into two categories:

1) Survey of ongoing research elsewhere:

The first workshop in June 1982 gave a broad overview of algorithms and data structures for computational geometry used in many different applications.
The second workshop concentrated on sweep algorithms and data structures for CAD, reflecting more closely our own research interests.

2) Our own research:

G. Beretta has implemented the plane-sweep algorithm, as described in the quarterly report of December 1982, on the personal computer Lilith, developed at ETH Zürich.

M. Mäntylä (Helsinki University of Technology) and J. Nievergelt are developing space-sweep algorithms for solving intersection problems in 3-dimensional space. The space-sweep will be described in a later report.

K. Hinrichs has implemented the grid file on the Lilith personal computer and on a DEC VAX 11. On the Lilith there exists a graphic user interface as described in the paper included with this report.

H. Hinterberger is following up on his earlier work in concurrency control for the grid file (described in the September 82 quarterly report).

The grid file is now being used for analyzing photographic satellite data. We hope to report on this work later.

The grid file:
a data structure designed to support proximity queries on spatial objects

K. Hinrichs and J. Nievergelt
Institut für Informatik, ETH, CH-8092 Zürich

Abstract

We describe a technique for storing large sets of spatial objects so that proximity queries are handled efficiently as part of the accessing mechanism. This technique is based on a transformation of spatial objects into points in higher-dimensional spaces and on a data structure called the grid file. The grid file was designed to store higly dynamic sets of multi-dimensional data in such a way that it can be accessed using few disk accesses: a point query requires two disk accesses, a range query requires at most two disk accesses per data bucket retrieved. The efficiency of our technique is based on two facts:

1) many types of proximity queries lead to cone-shaped regions of the search space, and
2) the grid file allows an efficient enumeration of all the points in such a cone.

Contents

Paper presented at the
Workshop on Graphtheoretic Concepts in Computer Science
Osnabrück, June 1983

Supported in part by the European Research Office, US Army, under contract DAJA37-82-C-00589, R&D 4093-CC.

1. Limitations of conventional techniques for storing spatial objects

File structures for storing large amounts of geometric data are of increasing importance in applications such as computer graphics, geographic information systems, and computer-aided design, for example in VLSI. Such interactive applications of geometric data structures require storage schemes that support efficient *proximity queries* such as intersection, containment and range queries. In addition to proximity, it is desirable that the storage scheme reflects as many topological and geometric properties of spatial objects as conveniently possible, such as the type of an object, or bounds on its spatial extension.

Much effort has been invested in the development of data structures for managing sets of *points* in 1-, 2- and 3-dimensional space (e.g. binary trees, quad trees [BenFr79], oct trees and priority search trees [McCr82]). In contrast, known techniques for managing *spatial objects*, that is, geometric configurations that extend over regions of space, are primitive. Most often, data structures designed for storing points are adapted in a straightforward manner to storing spatial objects, as the following two widely used schemes show.

1) An object may be identified with a *set of representative points* that define some important aspects of the object - for example, the vertices of a polyhedron. Each of these points is stored together with a reference to the corresponding object. The resulting redundancy inherent in this storage scheme (the same object is being pointed at from many different places) makes updating expensive. For example, inserting an object requires inserting all of its representative points; since these may be spread out in space, they are likely to be spread out over the disk as well. Thus updating even a simple object (whose description easily fits onto a page) may require several disk accesses.

2) An object may be stored via *one representative point*, such as its center of gravity. The information that a given point in space is the representative point of a stored object conveys *no further information* (such as the type of object it is, or its extent) without accessing the full description of the object, which we assume is stored on disk. Therefore proximity-based access is poorly supported by this approach.

We present an alternative technique for storing spatial objects designed to overcome some of these deficiencies. It is based on a *transformation of spatial objects into points in higher-dimensional spaces* and on a data structure called the *grid file*. The presentation and assessment of this new approach is organized as follows:

Section 2 presents two well-known techniques used to approximate or decompose complex spatial objects into simpler ones. If these simple primitives (e.g. boxes, spheres and cylinders) are determined by a fixed number of parameters, then they may be considered to be points in higher-dimensional spaces.

Section 3 describes a geometric interpretation of basic access and proximity problems on sets of simple objects. Range, intersection and containment queries result in cone-shaped search regions in the corresponding higher-dimensional spaces.

Section 4 describes the grid file which is designed to sweep such search regions efficiently. This multi-key file structure treats all space dimensions symmetrically, adapts its shape to dynamically varying contents, and supports efficient range queries.

Section 5 describes the functional aspects of the implementation of the grid file and of a graphic user interface that turns the grid file into a simple geometric data base.

Section 6 evaluates the grid file in relation to other file structures for storing multi-dimensional data.

2. Representation of spatial objects as points in higher-dimensional spaces

Complex spatial objects can be represented or approximated by simpler ones in a variety of ways. Two widely used techniques are illustrated in Fig. 1 and Fig. 2. In constructive solid modelling an object is represented as a boolean expression over simpler objects (Fig. 1). In other cases it is approximated by enclosing it in a container chosen from a class of simple shapes (Fig. 2). The most important properties for proximity-based access to spatial objects are preserved by such representations.

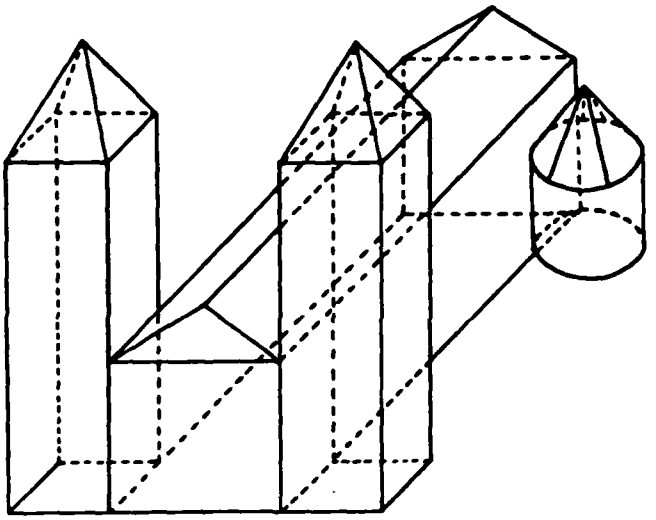


Fig. 1: Decomposition of complex objects into simpler ones.

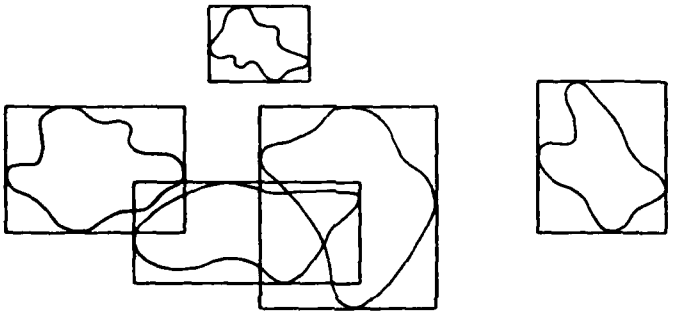


Fig. 2: Approximation of complex objects by containers.

As building blocks of such representations or approximations we consider classes of simple spatial objects, such as the class of all aligned rectangles in 2-d space, or the class of all spheres in 3-d space. Within its class, each object is defined by a small fixed number of parameters. For example, an aligned rectangle (i. e. a rectangle with sides parallel to the axes) can be described by its center (cx, cy) and the half-length of each side, dx and dy; and a sphere by the coordinates of its center (cx, cy, cz) and its radius r.

Any simple object, defined within its class by k parameters, can be considered to be a point in a k-dimensional space assigned to its class. *All of its geometric and topological properties can be deduced from the class it belongs to and from the coordinates of the corresponding point.* For example, an aligned rectangle becomes a point in 4-dimensional space, a rectangle in general position is represented by a point in 5-dimensional space.

The technique to be presented for storing spatial objects works for any choice of parameters by which simple objects can be represented. However, depending on characteristics of the data to be processed, some choices of parameters are better than others. Let us discuss some considerations that may determine the choice of parameters.

1) Distinction between *location parameters* and *extension parameters*. For some classes of simple objects it is reasonable to distinguish location parameters, such as the center (cx, cy) of an aligned rectangle, or the center (cx, cy, cz) of a sphere, from extension parameters, such as the half-sides dx and dy, or the radius r. This distinction is always possible for objects that can be described as Cartesian products of spheres of various dimensions. For example, a rectangle is the product of two 1-dimensional spheres, a cylinder the product of a 1-d and a 2-d sphere. Whenever this distinction can be made, cone-shaped search regions generated by proximity queries as described in section 3 have a simpler intuitive interpretation: The subspace of the location parameters acts as a "mirror" that reflects a query point.

2) Independence of parameters, uniform distribution. As an example, consider the class of all intervals on a straight line (Fig. 3a). If intervals are represented by their left and right endpoints, lx and rx, the constraint lx <= rx restricts all representations of these intervals by points (lx, rx) to the triangle above the diagonal in Fig. 3b. Any data structure that organizes the embedding space of the data points, as opposed to the particular set of points that must be stored, will pay some overhead for representing the unpopulated *half of the embedding space*. A coordinate transformation that distributes data all over the embedding space will lead to more efficient storage. The situation can be even worse than than this. In most applications the building blocks from which complex objects are built are much smaller than the space in which they are embedded, as the size of a brick is small compared to the size of a house. If so, parameters that locate boundaries of an object, such as (lx, rx), are highly dependent on each other. Fig. 3b shows how short intervals on a long line cluster along the diagonal, leaving large regions of a large embedding space unpopulated; whereas the same set of intervals, represented in Fig. 3c by separating location parameters from extension parameters, fills a smaller embedding space in a much more uniform way. The data distribution of Fig. 3c is easier to handle than the one of Fig. 3b.

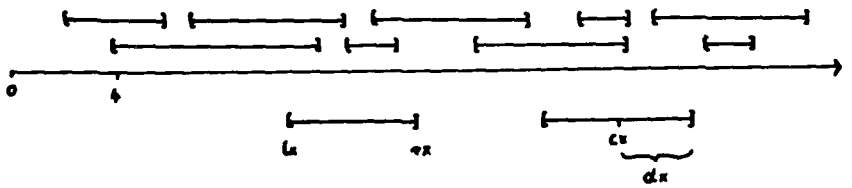


Fig. 3a: Intervals on a straight line.

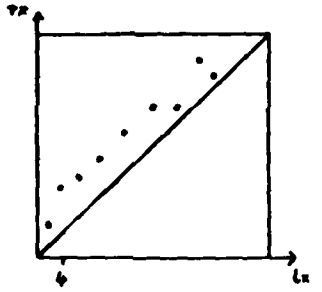


Fig. 3b: Representation of intervals by left and right endpoints.

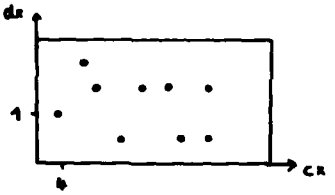


Fig. 3c: Representation of intervals by center and half length.

3. Proximity queries lead to cone-shaped search regions

The example of intersection of simple objects will show how to treat basic proximity queries. Given a class C of simple spatial objects defined by a fixed number of parameters, the corresponding higher-dimensional space H, and a set S c C of simple objects represented as points in H, we consider two types of queries:

- point query: given a query point q, find all objects in S which contain q;
- point set query: given a set Q of points, find all objects in S which intersect Q.

Given a point q we can describe *exactly* the region in H that contains all points representing objects in S which contain q. For instance let C be the class of intervals on a straight line. An interval given by its center cx and its half length dx contains a point q with coordinate qx if and only if cx - dx <= qx <= cx + dx (Fig. 4a). Fig. 4b shows the cone-shaped region in 2-d space that contains all points representing intervals which contain q.

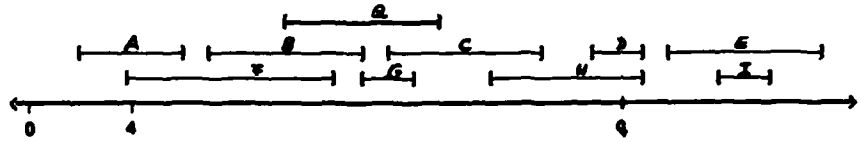


Fig. 4a: Point queries and point set queries in the class of intervals on a straight line (q = query point, Q = query interval).

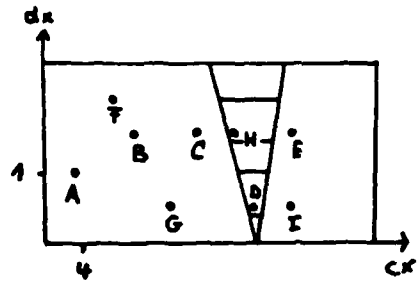


Fig. 4b: Search region for a point query in the class of intervals on a straight line.

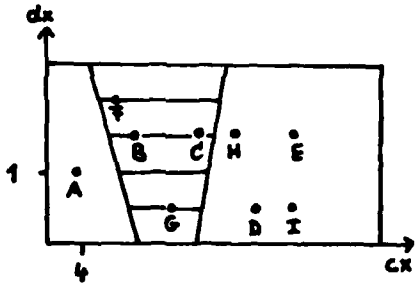


Fig. 4c: Search region for an interval intersection query.

Given a set of points Q, the region in H that contains all objects in S which intersect Q is the *union* of the regions that result from the point queries for each point in Q. This *union of cones* is a particularly simple region in H if the query set Q is a simple spatial object. Consider three examples.

1) The class of intervals along the x-axis (Fig. 4a). An interval (cx, dx) intersects a query interval Q = (cq, dq) iff its representing point lies in the shaded region shown in Fig. 4c; this region is given by the inequalities cx - dx <= cq + dq and cx + dx >= cq - dq.

2) The class C of aligned rectangles in the plane (with parameters as in section 2) can be treated as the Cartesian product of two classes of intervals, one along the x-axis, the other along the y-axis. All rectangles which intersect a given rectangle Q (Fig. 5a) are represented by points in 4-d space lying in the Cartesian product of two interval intersection query regions (Fig. 5b). The region is shown by its projections into the cx-dx-plane and the cy-dy-plane.

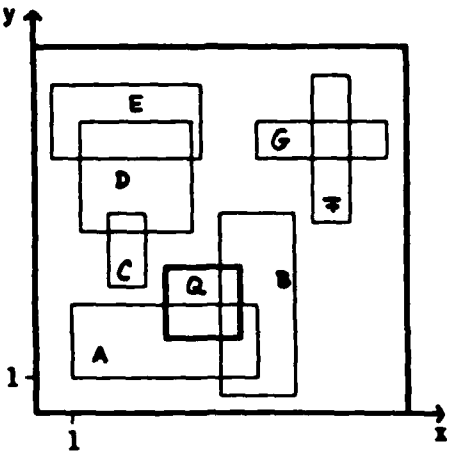


Fig. 5a: Aligned rectangle intersection (Q = query rectangle).

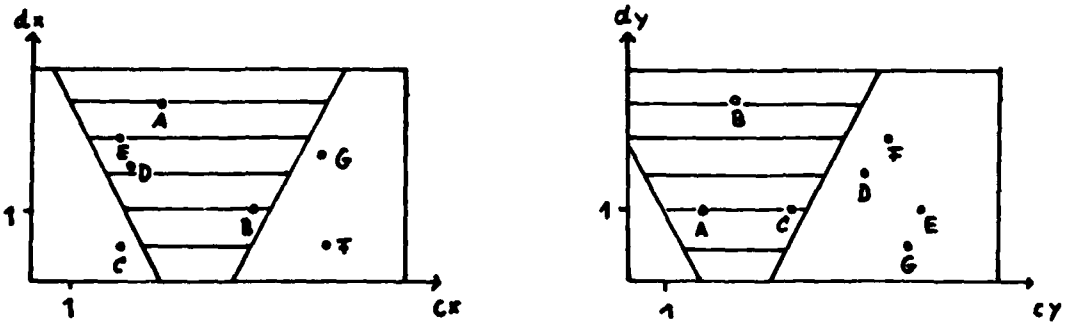


Fig. 5b: Search region for a rectangle intersection query.

3) The class C of circles in the plane. As parameters for the representation of a circle as a point in 3-d space we choose the coordinates of its center (cx, cy) and its radius r. All circles which contain a point q (Fig. 6a) are represented in the corresponding 3-d space by points lying in the cone shown in Fig. 6b. The axis of the cone is parallel to the r-axis, its vertex is q considered as a point in the cx-cy-plane. All circles which intersect a line segment L are represented by points lying in the cone-shaped solid shown in Fig. 6c. This solid is obtained by embedding L in the cx-cy-plane and moving the cone of Fig. 6b along L. All circles which intersect a rectangle R are represented by points lying in the cone-shaped solid shown in Fig. 6d.

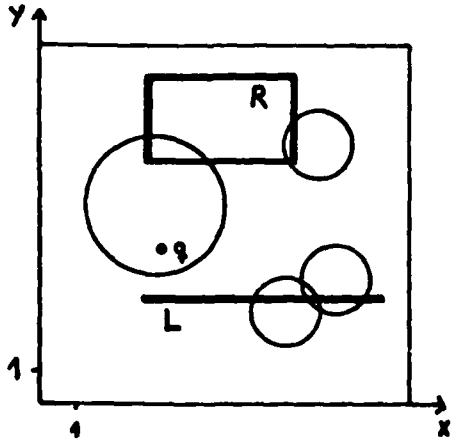


Fig. 6a: Point queries and point set queries in the class of circles in the plane (q = query point, L = query line segment, R = query rectangle).

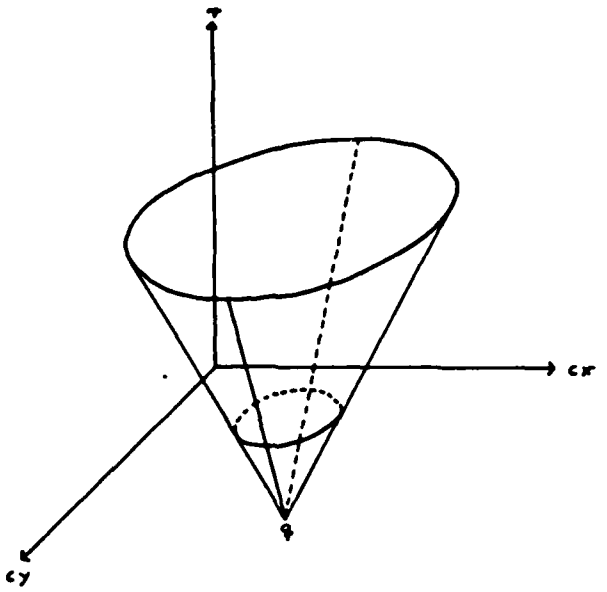


Fig. 6b: Search region for a point query in the class of circles in the plane.

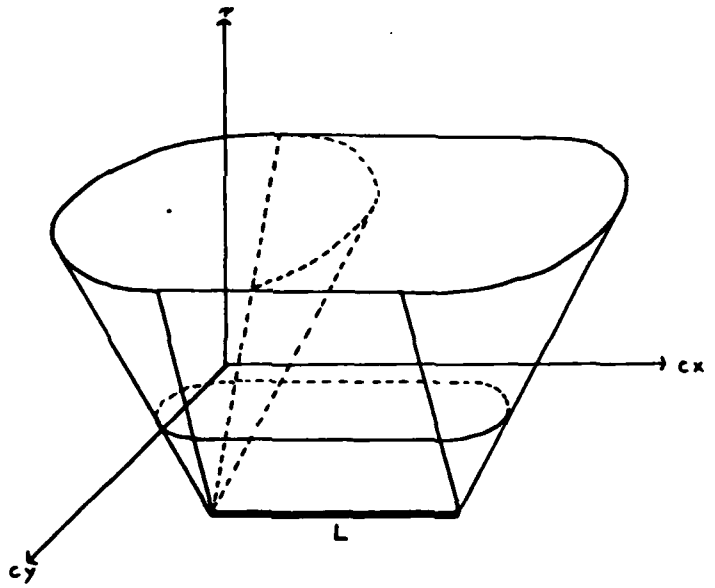


Fig. 6c: Search region for an intersection query between a given line segment L and circles.

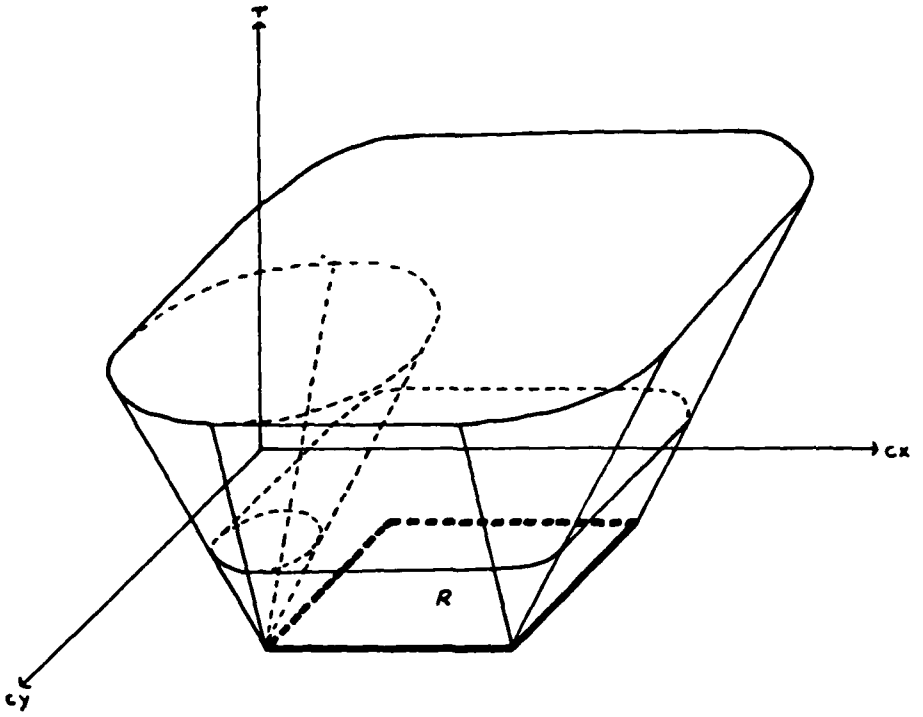


Fig. 6d: Search region for an intersection query between a given rectangle R and circles.

Although the examples above involve only 1-d and 2-d objects, analogous proximity queries on sets of 3-dimensional simple spatial objects lead to cone-shaped search regions in higher-dimensional spaces in exactly the same way.

4. The grid file as a data structure for storing simple spatial objects

The efficiency of a file structure for storing simple spatial objects depends on how fast basic proximity queries can be answered. For typical applications envisioned, such as CAD, the size of the geometric data bases involved is such as to preclude the possibility of keeping all the objects being processed in central memory. Instead, they must be fetched from disk during processing, often in response to proximity queries. Thus speed of access is best measured in terms of the number of disk accesses involved.

As in the case of object intersection, each of these queries defines for each type of object a cone-shaped search region in the corresponding higher-dimensional space. To minimize the number of disk accesses in range queries an efficient file structure should preserve locality: objects of the same type which are represented by points that are near to each other should have a high probability of being stored in the same physical disk block. Furthermore the file structure should handle all space dimensions symmetrically and adapt its shape dynamically under insertions and deletions.

The grid file, a file structure for storing multi-dimensional data, was designed to meet these requirements. It partitions the data space from which the points are drawn according to an orthogonal grid. The grid on a k-dimensional data space is defined by k 1-dimensional arrays, called the scales. Each element of a scale represents a (k-1)-dimensional hyperplane that partitions the space into two. There is a one-to-one correspondence between the grid defined by the scales and a k-dimensional dynamic array, called the grid directory. An element of this array is a pointer to a disk block, called a data bucket, which contains all data points that lie in the corresponding grid cell. To avoid low bucket occupancy, several grid cells may share a bucket. Such a set of grid cells is called a bucket region. Bucket regions are only allowed to have the shape of a k-dimensional rectangular box. These bucket regions are pairwise disjoint, together they span the data space. Fig. 7 shows the organization scheme of the grid file.

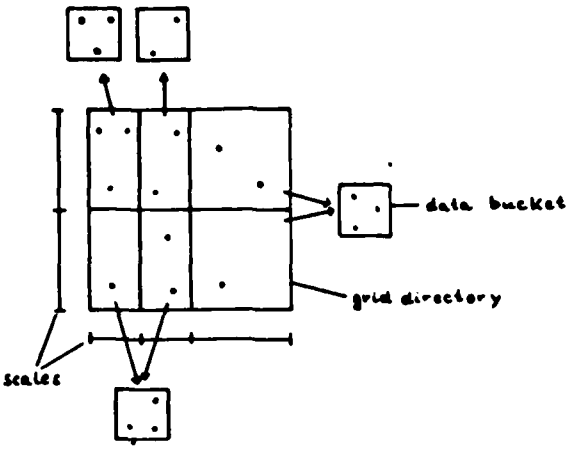


Fig. 7: Organization scheme of the grid file (k = 2).

During operation of the file system the grid directory and the scales need to be modified in response to insertions and deletions of points. An overflowing bucket is split into two by the following rule: if its bucket region covers only one grid cell the grid has to be extended by a (k-1)-dimensional hyperplane that cuts the bucket region into two. This is achieved by inserting a new boundary into one of the scales and maintaining the one-to-one correspondence between the grid and the grid directory. In any case there now exists a hyperplane separating the bucket region into two. To each of these two regions a new bucket is attached by updating the grid directory, and the points stored in the overflowing bucket are distributed among the new buckets correspondingly (Fig. 8).

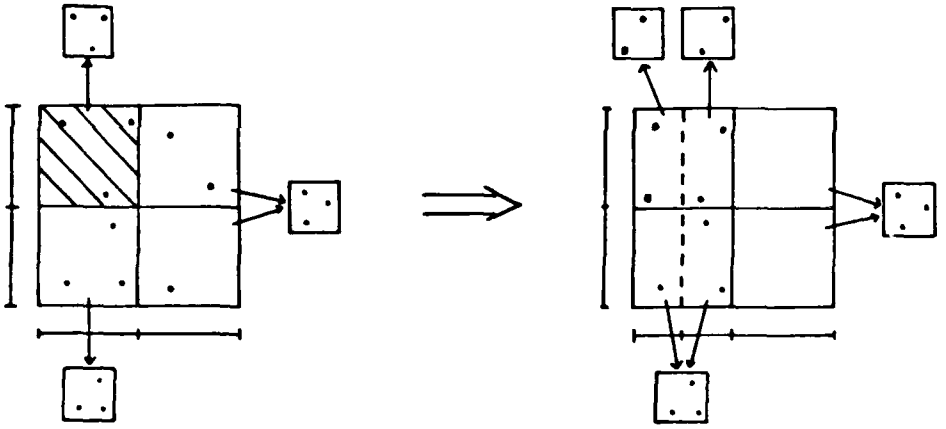


Fig. 8: Splitting of a bucket (k = 2).

Conversely, if the joint occupancy of two adjacent buckets drops below a certain threshold, they may be merged into one, as long as the new bucket region remains rectangular.

The grid directory is likely to be large and must therefore be kept on disk, but the linear scales are small and can be kept in central memory. Therefore the grid file realizes the *two-disk-access principle* for single point retrieval: by searching the scales, the k coordinates of a point are converted into interval indices without any disk accesses; these indices provide direct access to the correct element of the grid directory on disk, where the bucket address is located. In a second access the *correct data bucket* (i.e the bucket that contains the point to be searched for, if it exists) is read from disk.

In most data structures, in particular those based on pointer chains or on overflow buckets, an unsuccessful search takes considerably longer than an average successful search. The grid file is a notable exception, in that *both successful and unsuccessful single point queries are handled in two disk accesses.*

We have seen that basic proximity queries on sets of simple spatial objects represented as points in higher-dimensional spaces lead to cone-shaped search regions. In order to answer such a query with a grid file, all grid cells which intersect the search region must be computed, since the buckets corresponding to these grid cells are exactly those that can contain points to be searched for (Fig. 9).

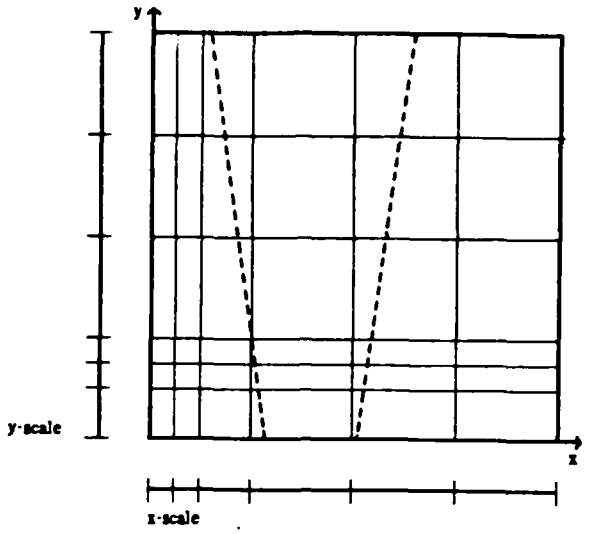


Fig. 9: Basic proximity query in a grid file.

This computation is done with the aid of the scales *without any disk accesses*. The buckets corresponding to the intersecting grid cells are then accessed via the grid directory. If a bucket region is not completely contained in the search region, each point stored in the bucket must be examined individually to determine whether it belongs to the search region.

The grid directory is again a large amount of multi-dimensional data that must be stored on disk, and a cone-shaped search region in the space of data points results in a "staircase approximation" to this cone in the directory. As in the case of data points, grid directory elements which correspond to contiguous grid cells should have a high probability of being stored in the same disk block. Thus, in order to "minimize" the number of disk accesses, the grid directory is again stored on disk as a grid file. The resulting data compression leads to the concept of the *resident grid directory*: it manages the grid directory on disk but is small enough to be kept in central memory, like the scales. The resident grid directory is a scaled down version of the real one, in which the limit of resolution is coarser. It distributes grid directory elements among disk blocks as shown in Fig. 10. All space dimensions are handled symmetrically, as opposed to the common row by row or column by column storage schemes for arrays.

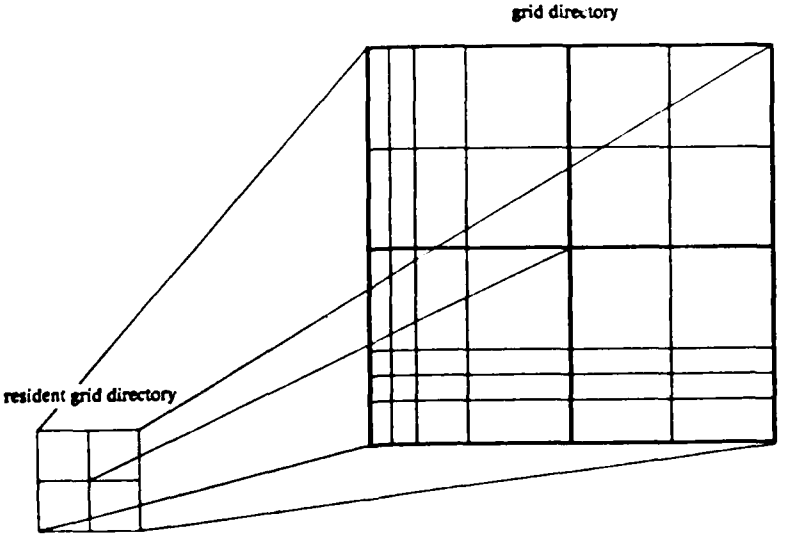


Fig. 10: The resident grid directory.

A more detailed description of the grid file, along with performance measurements, can be found in [NHS 81].

5. Implementation of the grid file

The grid file used for storing geometric data has been implemented in MODULA-2 on the *Lilith* personal computer [Wir 81] and the DEC VAX 11/780. On the Lilith a graphical user interface exists (see Fig. 11), embedded in the integrated interactive operating system XS-1 [Ber 82]. It allows the user to input geometric data (rectangles, circles and line segments) and queries (intersection, containment) interactively with a mouse. On the VAX the grid file runs under VMS. An experiment is underway to study clustering algorithms that use the grid file for preprocessing photographic satellite data.

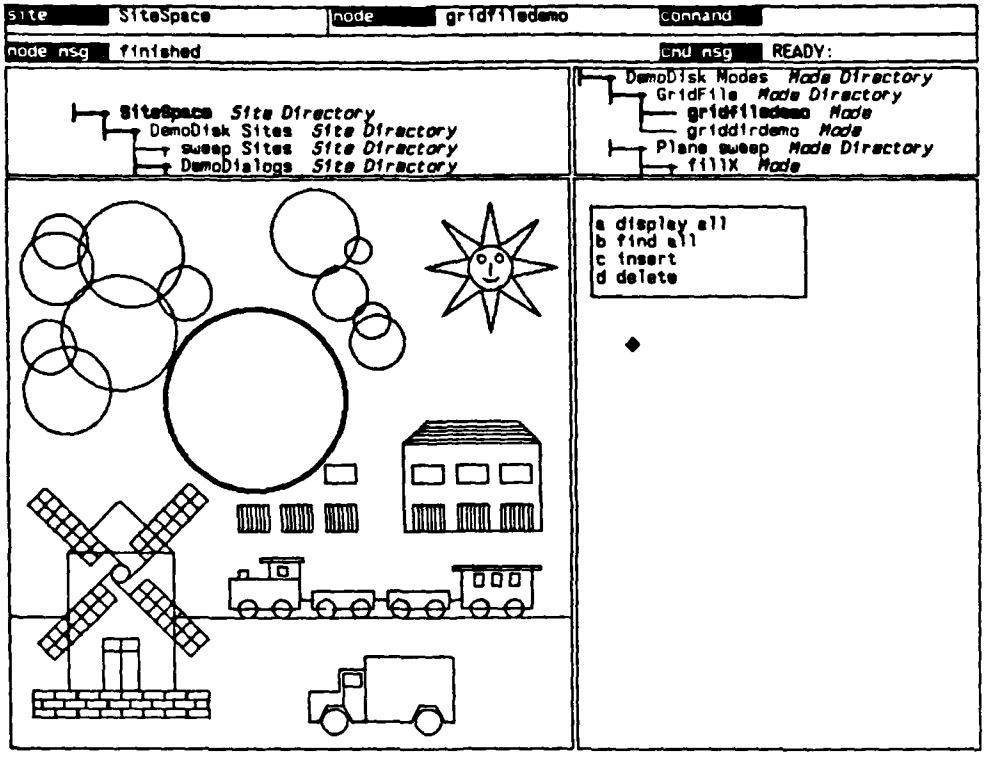


Fig. 11: Graphical user interface to the grid file in the XS-1 system, showing the result to the query: "Find all objects that do NOT intersect a given query circle".

In order to facilitate the portability of the grid file program, we have defined two interfaces: One towards the host (hardware and operating system), the other towards client programs.

The module "HOST" separates the machine- and disk-dependent parts from the grid file module itself. It has to provide procedures for:

- creating and initializing the disk storage;
- opening and closing communication channels between the disk and the grid file module;
- creating, deleting, reading and writing disk blocks;
- managing empty disk blocks;
- allocating and deallocating dynamic storage.

Therefore the grid file can be transferred to other computers by only adapting this lower module.

In the current implementation of the module "CLIENT" the keys of the records are only allowed to be of type *cardinal*. In a later version other key types such as reals or character strings will be supported. Besides their identifying keys x1, ... , xk, records may contain some additional information which is not of interest for the grid file. When creating a grid file the user has to declare whether the records are uniquely identified by their keys. "CLIENT" provides procedures for :

- creating, deleting, opening and closing a gridfile;
- inserting and deleting records in a grid file;
- changing non-key information in a record.

Furthermore "CLIENT" allows the following queries to be performed on the data:

- identity query: find all records with given key values x1, .., xk (if keys are unique at most one record will be found);
- key range query: find all records whose key values xi lie in given intervals [li, ui] (1 <= i <= k);
- procedure query: the user has to write a procedure which is passed to the grid file module and which determines whether a grid cell (given by intervals [li, ui] (1 <= i <= k)) contains records to be searched for; this gives the user the possibility to influence the query during execution;
- nextabove, nextbelow: given key i with key value xi, find the records with key values above or below xi and next to xi; this gives the user the possibility to read all the records sequentially with respect to one key.

6. Comparison of the grid file with other file structures

Having seen that object storage, when reduced to multi-dimensional point storage, leads to cone-shaped search regions, the following comparison of the grid file with conventional data structures can be limited to point storage. As stated in section 4, an efficient file structure for storing multi-dimensional data must preserve neighborhood relations. Among the structures that preserve proximity to various degrees, we consider inverted files, multi-dimensional trees [Ben 79], and interpolation-based index maintenance [Bur 82].

The *inverted file* treats space dimensions in a highly asymmetric way: space is partitioned into thin slices orthogonal to the primary key axis (Fig. 12).

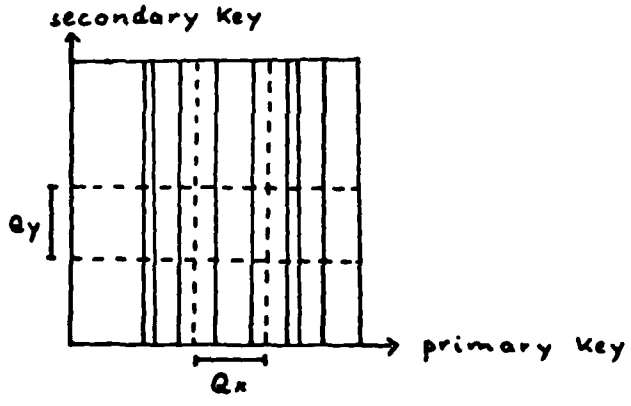


Fig. 12: Space partitioning by an inverted file.

All data points in a slice are stored in one data bucket. As Fig. 12 shows, a range query Qx orthogonal to the primary key is handled very efficiently (with "high precision" in the terminology of information retrieval), whereas range queries on secondary keys, such as Qy, or cone-shaped queries, yield results of low precision. "Salami slices" are poor primitives for approximating linearly bounded search regions.

Like the grid file, a *multi-dimensional tree* handles all space dimensions symmetrically. Thus the space partition induced yields rectangular bucket regions better suited for approximating cone-shaped search regions than the slices of the inverted file.

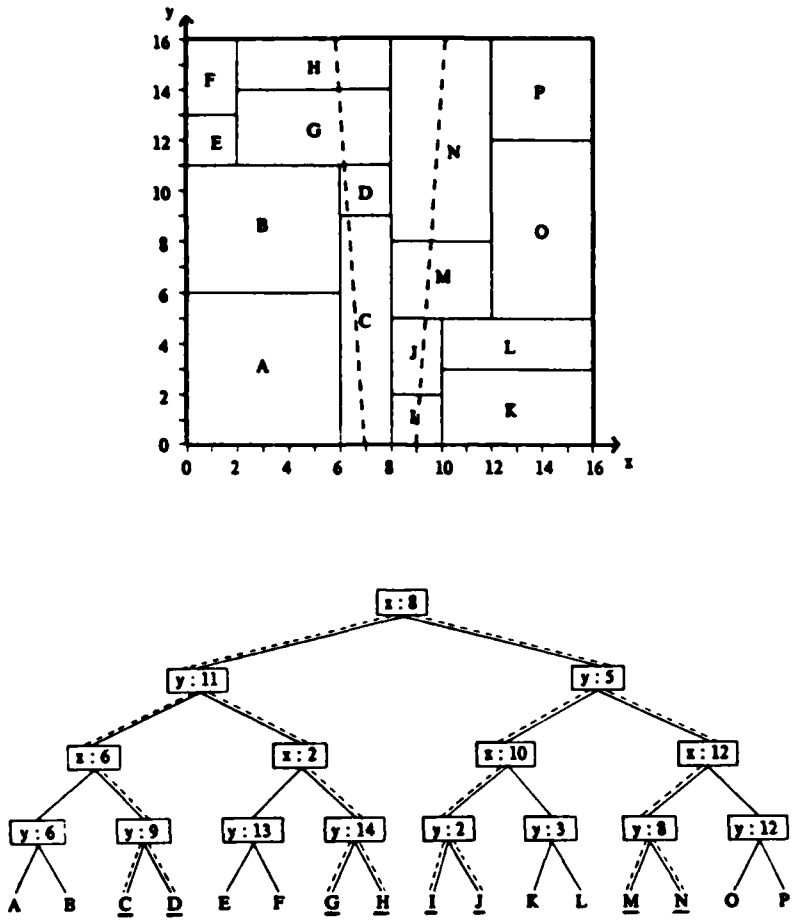


Fig. 13: Space partitioning by a multi-dimensional tree.

The difference between the grid file and the multi-dimensional tree shows up in computing the buckets whose regions intersect the query region. The information about the boundaries of bucket regions is distributed across the entire tree, and in general many root-to-leaf paths must be traversed before a given bucket region can be included or excluded from the answer. The tree is likely to be large and therefore has to be stored on disk. Some of the paths will share links, but contiguous bucket regions in general correspond to non-contiguous subtrees (Fig. 13), so even a small query region is likely to be spread across the tree, and may require many disk accesses to retrieve all the relevant data buckets.

Interpolation-based index maintenance, recently introduced by Burkhard, reflects the current trend in designing file structures: Away from list structures that must be broken across disk block boundaries, towards address computation techniques that approximate direct access [Nie 81]. It uses a grid partition of the search space, at intervals determined by a radix, similarly to the grid file. The correspondence between bucket regions and buckets is given by formulas ("interpolations") rather than by a directory as used in the grid file. The trade-offs involved in the decision of using a directory, as in the grid file, or avoiding it, as in interpolation-based index maintenance, are an interesting topic for research.

Based on experience with grid file implementations we believe that the approach presented in this paper is effective for storing large sets of spatial objects in such a way as to support proximity queries. The considerations above suggest that it is a promising alternative to conventional techniques. We hope others will join us in experimenting with the approach of transforming objects into points in higher-dimensional spaces, and storing them in a grid file.

References

[Ben 79]
J. L. Bentley: Multi-dimensional Binary Search Trees in Database-Applications,
IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, 1979, 333 - 340.

[BenFr 79]
J. L. Bentley, J. H. Friedman: Data Structures for Range Searching,
Computing Surveys, Vol. 11, No. 4, 1979, 397 - 409.

[Ber 82]
G. Beretta, et al: XS-1: An integrated interactive system and its kernel,
Proc. 6-th Int. Conf. Software Engineering, Tokyo, 340-349, IEEE Computer Society, 1982.

[Bur 82]
W. A. Burkhard: Interpolation-Based Index Maintenance,
T. R. CS-053, UCSD, Dep. of EECS, 1982, to appear in BIT.

[Enc 82]
J. Encarnacao, F.-L. Krause (editors): Proceedings of the IFIP WG 5.2 Working
Conference on File Structures and Data Bases for CAD, 1982.

[McCr 82]
E. M. McCreight: Priority Search Trees, Report CSL-81-5, XEROX Corp., 1982.

[NHS 81]
J. Nievergelt, H. Hinterberger, K. C. Sevcik:
The grid file: an adaptable, symmetric multi-key file structure,
Report No. 46, Institut für Informatik, ETH Zürich, 1981 (to appear in ACM TODS).

[Nie 81]
J. Nievergelt: Trees as data and file structures,
in CAAP '81, Proc. 6-th Coll. on Trees in Algebra and Programming,
E. Astesiano and C. Bohm (eds.), Lecture Notes in Computer Science 112, 35-45,
Springer Verlag 1981.

[Wir 81]
N. Wirth: The personal computer Lilith,
Proc. 5th International Conference on Software Engineering, 2-15,
IEEE Computer Society Press, 1981.


ATE
LME